Context-Sensitive Staged Static Taint Analysis for C with LLVM

Xavier N. Noumbissi
xnoumbis@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

Problem: Prevent Software Vulnerabilities

- Format String Attacks
- SQL Injection
- Cross Site Scripting, etc.

OpenSSL Security Bug

- Heartbleed (April 7, 2014)
- Code uses user provided buffer length without checking real buffer size
- Vulnerability gives access to server's private key
- Could be detected by static analysis

Heartbleed in OpenSSL[1]

1. **byte_swapping:** Performing a byte swapping operation on `p` implies that it came from an external source, and is therefore tainted.
2. **var_assign_var:** Assigning: `payload = ((unsigned int)p[0] << 8) | (unsigned int)p[1]`. Both are now tainted.

```
2446        n2s(p, payload);
2447        pl = p;
2448
```

3. Condition `s->msg_callback`, taking true branch

```
2449        if (s->msg_callback)
2450                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2451                        &s->s3->rrec.data[0], s->s3->rrec.length,
2452                        s, s->msg_callback_arg);
2453
```

4. Condition `hbtype == 1`, taking true branch

```
2454        if (hbtype == TLS1_HB_REQUEST)
2455                {
2456                unsigned char *buffer, *bp;
2457                int r;
2458
2459                /* Allocate memory for the response, size is 1 bytes
2460                 * message type, plus 2 bytes payload length, plus
2461                 * payload, plus padding
2462                 */
2463                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
2464                bp = buffer;
2465
2466                /* Enter response type, length and copy payload */
2467                *bp++ = TLS1_HB_RESPONSE;
2468                s2n(payload, bp);
```

CID 1201699 (#1 of 1): Untrusted value as argument (TAINTED_SCALAR)

5. **tainted_data:** Passing tainted variable `payload` to a tainted sink.

```
2469                memcpy(bp, pl, payload);
```

[1] Image from Andy Chou's blog at Coverity

Taint Analysis

- Tracks usage of untrusted program input
- Untrusted program input: Tainted Input
- Taint source: tainted input origin (e.g. system call return values)
- Taint sink: use of tainted input

Taint Analysis: taint propagation

- Taint propagation: operations depending on tainted input generate tainted values
- Explicit taint propagation: data flow
- Implicit taint propagation: control flow

Example

```
1  int main() {
2    int x, b1, b2, y;
3    scanf("%d", &x);
4    b1 = even(x);
5    b2 = odd(3);
6    y = compute(x);
7    return 0;
8  }

10 int compute(int x) {
11   int sum, i;
12   if (x == 2)
13     scanf("%d", &sum);
14   else
15     sum = 0;
16   for(i = 0; i < x; ++ i)
17     sum += i;
18   return sum;
19 }

21 int odd(int x) {
22   if (x == 1)
23     return 0;
24   else
25     return even(x - 1);
26 }

28 int even(int x) {
29   if (x == 0)
30     return 1;
31   else
32     return odd(x - 1);
33 }
```

Taint Information

+ Line 3: x is tainted
+ Line 6: y may be tainted (needs interprocedural analysis)
+ Line 13: sum is tainted

Contributions

- Algorithm to statically detect tainted values' flow in C programs
- Handling of interprocedural taint propagation
- WAINT: Implementation of the algorithm in LLVM

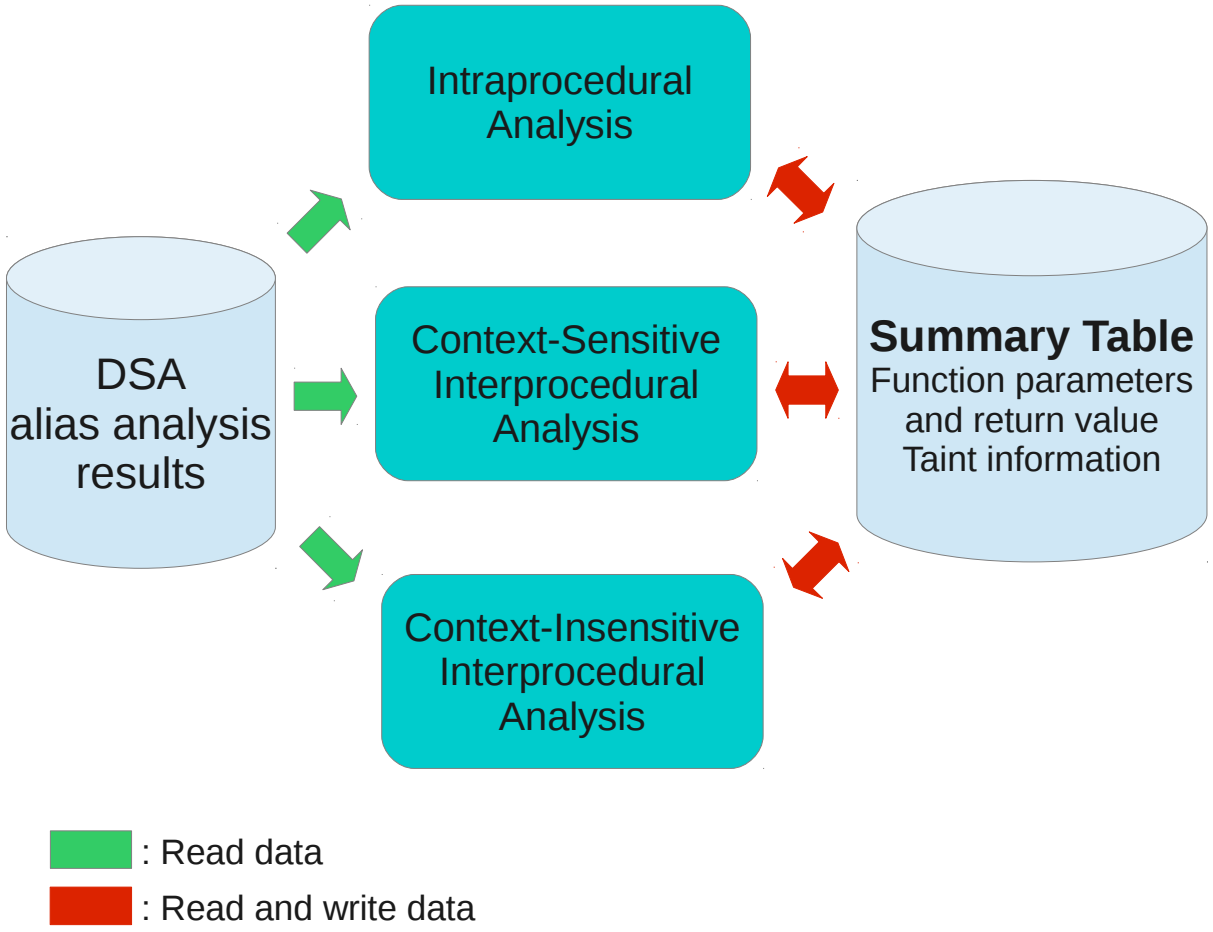
Source/Sink Specification

- Developer specify sources and sinks in configuration file
- Analysis do not analyze sources and sinks
- Analysis use annotations for sources: taint propagation (from configuration file)

WAINT Analysis

- Summary table to store function parameters and return value taint information
- Intraprocedural analysis: discovery of taint sources, initial values for summary table
- Context-Sensitive analysis: interprocedural taint tracking using DSA alias analysis
- Context-Insensitive analysis: use summary table information

WAINT Analysis (Flow)



Intraprocedural Analysis

Statement Type	C Code
COPY	$p = q$
LOAD	$p = *q$
STORE	$*p = q$
CALL	*call func*

Intraprocedural Analysis: transfer functions

- COPY [$p = q$]: taint p iff q is tainted
- LOAD [$p = *q$]: taint p iff $t_q = *q \wedge t_q$ is tainted
- STORE [$*p = q$]: taint $t_p = *p$ iff q is tainted
- CALL [*call func*(p)]: taint all t_p `s.t` $t_p = *p$

Example (2)

```
1  int main() {
2    int x, b1, b2, y;
3    scanf("%d", &x);
4    b1 = even(x);
5    b2 = odd(3);
6    y = compute(x);
7    return 0;
8  }

10 int compute(int x) {
11   int sum, i;
12   if (x == 2)
13     scanf("%d", &sum);
14   else
15     sum = 0;
16   for(i = 0; i < x; ++ i)
17     sum += i;
18   return sum;
19 }

21 int odd(int x) {
22   if (x == 1)
23     return 0;
24   else
25     return even(x - 1);
26 }

28 int even(int x) {
29   if (x == 0)
30     return 1;
31   else
32     return odd(x - 1);
33 }
```

Taint Information

+ Line 3: x is tainted
+ Line 6: y may be tainted (needs interprocedural analysis)
+ Line 13: sum is tainted

Summary Table after Intraprodural Analysis

Functions	Variables
even	x^u, ret^u
odd	x^u, ret^u
compute	x^u, ret^t
main	ret^u

Context-Sensitive Analysis

- Same transfer functions as intraprocedural analysis except CALL
- Use Data Structure Analysis (DSA): field- and context-sensitive alias analysis[2]
- Analysis of a callee start with taint assumptions from the caller
- Use summary table for procedure formals and return value initial taint information

[2] from LLVM creator Chris Lattner

Context-Insensitive Analysis

- Use functions' return value taint information from summary table
- In practice: useful after context-sensitive analysis

Example: WAINT

```
1  int main() {
2    int x, b1, b2, y;
3    scanf("%d", &x);
4    b1 = even(x);
5    b2 = odd(3);
6    y = compute(x);
7    return 0;
8  }

10 int compute(int x) {
11   int sum, i;
12   if (x == 2)
13     scanf("%d", &sum);
14   else
15     sum = 0;
16   for(i = 0; i < x; ++ i)
17     sum += i;
18   return sum;
19 }

21 int odd(int x) {
22   if (x == 1)
23     return 0;
24   else
25     return even(x - 1);
26 }

28 int even(int x) {
29   if (x == 0)
30     return 1;
31   else
32     return odd(x - 1);
33 }
```

Intraprocedural Analysis

+ **Line 3: x initially tainted**
+ **Line 13: sum initially tainted**
+ **Line 18: return value sum is tainted**
  - **compute() updates summary table**

Context-Sensitive Analysis

+ **Line 3: first parameter of even (x) is tainted**
+ **Line 6: first parameter of compute (x) is tainted**

Context-Insensitive Analysis

+ **Line 6: y is tainted (from summary table)**
  Intraprocedural analysis would not find this

: Initial taint information
: Existing taint information

Figure 1. Motivating Example

Current Implementation

Program	SLOC	Warnings	Runtime
Mongoose web server (4.1)	4*k*	36	140s
vlc-input (2.1.2)	16*k*	0	6s
Claws email client (3.9.3)	142*k*	219	11s
Apache web server (2.4.7)	144*k*	n/a	n/a (DSA crash)

TODOs/Future Work

- Handling of arrays, structs
- Handling of cycles (SCC) in call graph
- Investigate crash of DSA while running Apache
- Perform tests with other alias analysis

Conclusion

- **Hearbleed** bug in OpenSSL shows importance of taint analysis
- WAINT implements a context-sensitive taint analysis for C
- Preliminary results scale well up to $150k$ lines of code